Reporte de Análisis de Algoritmos de Ordenamiento

Hoja de Trabajo #3

Estudiantes

- Joao Castillo - Carnet 25776
- Kenett Ortega - Carnet 25777

---

1. Metodología

Herramientas Utilizadas

Profiler: - **Java Flight Recorder (JFR)** - Profiler integrado en OpenJDK 21 - JFR es un sistema de profiling de bajo overhead diseñado para producción - Genera archivos `.jfr` que pueden analizarse con JDK Mission Control o la herramienta `jfr` - Configuración utilizada: perfil `profile` que captura: - Muestras de ejecución (execution samples) - Asignación de memoria (object allocation) - Eventos de garbage collection - Uso de CPU por thread - Stack traces detallados

Medición de Tiempo: - `System.nanoTime()` de Java para mediciones precisas de tiempo - Las mediciones se realizaron en nanosegundos y se convirtieron a milisegundos - Se ejecutaron las pruebas en paralelo usando `ExecutorService` con 10 threads - Archivo de profiling generado: `benchmark-profile.jfr` (662 KB)

Entorno de Pruebas: - Lenguaje: Java 21 (OpenJDK 21.0.8) - Build Tool: Maven - Sistema Operativo: Linux (Kernel 6.12.41) - Tamaños probados: 10, 100, 1,000, 5,000, 10,000, 30,000, 50,000, 100,000 elementos

Metodología de Benchmarking

1. **Generación de Datos:**
   - Se generaron números enteros aleatorios entre 0 y 10,000
   - Se guardaron en archivo `numbers.txt` para consistencia
2. **Escenarios de Prueba:**
   - **Caso Promedio:** Datos desordenados aleatoriamente
   - **Mejor Caso:** Datos ya ordenados ascendentemente
3. **Ejecución con Profiling:**
   - JFR se inició automáticamente al comenzar el benchmark
   - Cada algoritmo se ejecutó en un hilo separado para mediciones paralelas
   - Se midió el tiempo desde el inicio hasta la finalización del ordenamiento
   - Los resultados se exportaron a CSV para análisis

- JFR capturó 9,773 execution samples durante la ejecución completa

4. **Análisis del Profiler:**
   - El archivo JFR generado contiene stack traces detallados
   - Se pueden identificar hotspots (métodos que consumen más CPU)
   - Ejemplo de output del profiler:
     ```
     pool-1-thread-1: GnomeSort.sort() - STATE_RUNNABLE
     pool-1-thread-2: MergeSort.merge() - STATE_RUNNABLE
     pool-1-thread-3: QuickSort.partition() - STATE_RUNNABLE
     ```

---

2. Algoritmos Implementados

2.1 Gnome Sort

Complejidad de Tiempo: - Mejor caso: O(n) - cuando los datos ya están ordenados - Caso promedio: $O(n^2)$ - Peor caso: $O(n^2)$

Complejidad de Espacio: O(1) - ordenamiento in-place

Descripción: Algoritmo simple similar a Insertion Sort que mueve elementos hacia atrás hasta encontrar su posición correcta.

2.2 Merge Sort

Complejidad de Tiempo: - Mejor caso: O(n log n) - Caso promedio: O(n log n) - Peor caso: O(n log n)

Complejidad de Espacio: O(n) - requiere arreglos temporales

Descripción: Algoritmo divide y conquista que divide el arreglo recursivamente y luego fusiona las partes ordenadas.

2.3 Quick Sort (con Median-of-Three)

Complejidad de Tiempo: - Mejor caso: O(n log n) - Caso promedio: O(n log n) - Peor caso: $O(n^2)$ - mitigado con median-of-three

Complejidad de Espacio: O(log n) - por la recursión

Descripción: Algoritmo divide y conquista que selecciona un pivote y particiona el arreglo. Implementado con selección de pivote median-of-three para evitar el peor caso en datos ordenados.

2.4 Radix Sort

Complejidad de Tiempo: - Mejor caso: $O(d \times n)$ donde d es el número de dígitos - Caso promedio: $O(d \times n)$ - Peor caso: $O(d \times n)$

Complejidad de Espacio: $O(n + k)$ donde k es el rango de valores

Descripción: Algoritmo de ordenamiento no comparativo que ordena números dígito por dígito usando counting sort.

2.5 Insertion Sort

Complejidad de Tiempo: - Mejor caso: O(n) - cuando los datos ya están ordenados - Caso promedio: $O(n^2)$ - Peor caso: $O(n^2)$

Complejidad de Espacio: O(1) - ordenamiento in-place

Descripción: Algoritmo simple que construye el arreglo ordenado insertando elementos uno por uno en su posición correcta.

---

3. Resultados Experimentales

Observaciones Clave

1. **Rendimiento General:**
   - **RadixSort** fue consistentemente el más rápido en todos los tamaños
   - **MergeSort** y **QuickSort** mostraron rendimiento O(n log n) esperado
   - **GnomeSort** e **InsertionSort** se volvieron extremadamente lentos con datasets grandes
2. **Caso Promedio vs Mejor Caso:**
   - **InsertionSort** y **GnomeSort** mostraron O(n) en datos ordenados
   - **QuickSort** con median-of-three mantuvo buen rendimiento en datos ordenados
   - **RadixSort** y **MergeSort** mostraron rendimiento similar en ambos casos
3. **Escalabilidad:**
   - A 100,000 elementos:
     - RadixSort: 6-16 ms
     - MergeSort: 27-36 ms
     - QuickSort: 22-30 ms
     - InsertionSort: 2-24,739 ms (dependiendo del caso)
     - GnomeSort: 1-51,461 ms (dependiendo del caso)

Gráficas

Ver `performance_graph.png` y `theoretical_complexity.png` para visualización completa de los resultados.

---

4. Conclusiones

1. **RadixSort** es el algoritmo más eficiente para ordenar números enteros en el rango probado, con complejidad lineal O(n).

2. **MergeSort** y **QuickSort** (con median-of-three) son excelentes opciones de propósito general con garantía O(n log n).
3. **InsertionSort** y **GnomeSort** solo son prácticos para datasets pequeños (<1000 elementos), pero tienen ventajas en datos casi ordenados.
4. La selección de pivote **median-of-three** en QuickSort es crucial para evitar degradación $O(n^2)$ en datos ordenados.
5. El uso de **paralelización** en el benchmarking permitió obtener resultados más rápidamente sin afectar la precisión de las mediciones individuales.

---

5. Testing y Control de Versiones

Pruebas Unitarias (JUnit)

- Se implementó un patrón de **contract testing** usando una clase abstracta `SortAlgorithmContract`
- Todas las implementaciones de sort extienden esta clase y heredan los mismos tests
- Se probaron casos: lista vacía, elemento único, ordenado, reverso, duplicados, strings, etc.
- **Resultado:** 54 tests pasaron exitosamente

Control de Versiones (Git)

Se mantuvieron más de 3 commits durante el desarrollo: 1. Implementación inicial de clases y algoritmos 2. Adición de tests JUnit y corrección de bugs 3. Mejora de QuickSort y sistema de benchmarking paralelo

---

Anexos

- `benchmark_results.csv` - Datos crudos de todas las pruebas
- `performance_graph.png` - Gráfica de rendimiento experimental
- `theoretical_complexity.png` - Gráfica de complejidad teórica
- Código fuente en repositorio Git